This document describes the new features of TV Computer MOPS and BASIC in version 2.0 that have been introduced since version 1.3.

1. ERRORS IN MOPS AND BASIC

1. Expression evaluation has been re-written so that the type of an expression (string or numeric) is not decided until after all relational terms in the expression have been parsed. Thus PRINT "ABC"<"BC" is now valid.

2. Rounding has been added to PRINT USING.

3. The addition of a polygon function in the video driver (see below) has shown up two minor problems in the video driver. In pre-2.0 versions, when a line is drawn, both the start and the end pixels of the line are plotted. Thus if two joined lines are plotted (as when plotting a polygon) the intersecting pixel is plotted twice, which does not matter except that in XOR plotting mode this intersecting pixel is plotted in the wrong colour (the paper colour instead of the ink colour).

   The problem has been corrected in 2.0 by changing the line drawing so that the first pixel of each line is not plotted. Since the beam has to be turned on to draw a line, and turning the beam on causes the first pixel to be plotted, this will not normally have any effect, but it does allow polygons to be plotted in XOR mode.

   To allow a sequence of joined lines to be drawn from BASIC in XOR mode, the 'beam on' video function has been altered so that the point at the beam position is not re-plotted if the beam is already on when a 'beam on' function call is made.

4. The maximum length of input strings allowed in the INPUT statement has been changed from 251 to 254. This is also true of version 1. but was ommitted from the documentation.

5. The SOUND statement has been modified to prevent it corrupting BASIC'S EXT 5 and EXT 6 vectors. This problem was not discovered until after version 1.3 was produced; it can be fixed in version 1.3 however by patching the ROM addresses 2825h, 2845h and 2876h from 0h to 17h (ie. 2732 ROM number 2). Note that this also affects VT-DOS, which in version 1.0 uses EXT 6 to enter the BASIC CLI. VT-DOS 1.1 uses EXT 4 instead of EXT 6, thus allowing it to be used with earlier versions of BASIC (see VT-DOS 1.1 changes).

2. ENHANCEMENTS TO MOPS AND BASIC

This section details the changes that have been made to MOPS and BASIC 2.0 in the extra ROM space available. Many of these features are desired because other machines (such as the Commodore 16+) competing with the TV Computer have similar features. In general, these features, described below, offer similar functionalities to other machines, but the methods of using them may differ to make them more suitable for the TV Computer environment, operating system and BASIC syntax. In many cases the extra features will be found to be an improvement on those found on other machines.

The new features are generally completely compatible with previous versions. All MOPS functions and BASIC syntax remain unchanged. All published and unpublished variable address are unchanged, although some extra areas of RAM, not previously used, have been utilized (see below). One possible incompatibility is that the BASIC syntax does not allow variable names to start with BASIC keywords, although they can contain them. Since the extended BASIC has extra keywords, it is possible that variale names which were previously valid can no longer be used; the chances of this preventing an existing program from working have however been minimized by choosing fairly long and infrequently used names for the new keywords.

Copyright (1987) IS Systems Ltd.

1. The following new graphics commands have been added to BASIC's PLOT command.

PLOT RECTANGLE (h, v [,d [,a [,r]]])

Plots a rectangle. h and v are the length of the sides (h being the horizontal side when the angle is 0), d is the number of sides to draw (default 4), a is the angle in radians from the positive x-axis that the rectangle is plotted at (default 0) and r is the aspect ratio (where 0 <= r < 2, default 6/5). The plotting starts at the current beam position, and the beam is left at the last point plotted (ie. not moved if all sides are drawn). The current plotting mode and line style are used. The entire rectangle must fit on the screen. Note that the beam must be on in order for anything to be plotted.

PLOT POLYGON (l, n [,d [,a [,r]]])

Plots a regular polygon. l is the length of the sides and n is the number of sides of the polygon (where 0 <= n <= 20). d is the number of sides to actually draw (default n), a is the angle in radians from the positive x-axis that the first side is plotted at (default 0), and r is the aspect ratio (where 0 <= r < 2, default 6/5). The plotting starts at the current beam position, and the beam is left at the end of the last side plotted (ie. not moved if all sides are drawn). The current plotting mode and line style are used. The entire polygon must fit on the screen. Note that the beam must be on in order for anything to be plotted.

PLOT ELLIPSE (h [,v [,b [,e [,a [,r]]]]])

Plots arcs of ellipses. h and v are the lengths of the semi-horizontal and semi-vertical axis respectively when the angle is 0, and if v is not given then it defaults to h. b and e are the begin and end angles of the arc in radians from the positive x-axis (default 0, so a complete ellipse is drawn). a is the angle in radians from the positive x-axis that the ellipse is drawn at (default 0) and r is the aspect ratio (where 0 <= r < 2, default 6/5). The centre of the ellipse is the current beam position, which is not affected. The current plotting mode is used. The entire ellipse need not necessarily fit on the screen. Note that the ellipse is always drawn, even if the beam is off.

Because of the limited arithmetic accuracy to which the ellipse drawing routine works internally, some ellipses may not be drawn quite as expected. In general, long, thin ellipses drawn at angles close but not equal to 0, pi/4, pi/2 etc. may be smaller than expected, and in extreme cases may not even be elliptical. To improve the internal arithmetic accuracy would have slowed down the ellipse drawing and would make the ellipse routine prohibitively large for the ROM space available.

To support these new graphics statements, the following new video driver functions have been added. Note that the rectangle drawing is not implemented as a video driver function since it is so simple.

@POLY (video function code 00001101B)

Input: DE -> input parameters.

Output: A = error code, 0 if successful.

This function draws polygons. On input, DE points to a buffer that contains the polygon parameters, as follows:

- 1 byte - Aspect ratio, 0 to 7Fh, 40h => 1:1.
- 1 byte - number of sides opolygon, 0 to 20.
- 1 byte - number of sides to draw, 0 to 20.
- 2 bytes - signed x offset from current beam position to end of first side.
- 2 bytes - signed y offset from current beam position to end of first side.

@ELLIP (video function code 00001110B)

Input: DE -> input parameters.

Output: A = error code, 0 if successful.

This function draws arcs of ellipses. On input, DE points to a buffer that contains the ellipse parameters. To save the ellipse function from having to calculate various trigonometric functions, some of the parameters must be pre-calculated. The parameters are:

```
2 bytes - SQR(R) ...................................... Q
2 bytes - (-P)/Q
4 bytes - (v^2) * (SIN(a)^2) + (h^2) * (COS(a)^2) ... R
4 bytes - (v^2) * (COS(a)^2) + (h^2) * (SIN(a)^2)
4 bytes - SIN(a) * COS(a) * ((v^2) - (h^2)) ......... P
1 byte  - 127*COS(s)
1 byte  - 127*SIN(s)
1 byte  - 127*COS(e)
1 byte  - 127*SIN(e)
1 byte  - Aspect ratio, 0 to 7Fh, 40h => 1:1
```

where

a is the angle of the ellipse.
h is the semi-horizontal axis length (when a=0).
v is the semi-vertical axis length (when a=0).
s is the angle of the start of the arc.
e is the angle of the end of the arc.

2. Relative plotting has been added to BASIC, as supported by all versions of the video driver. Relative co-ordinates are specified in the PLOT statement in the same way as absolute co-ordinates, but with a + or - symbol in front of them. For example:

```
PLOT +80,+0; +0,+80; -80,+0; +0, -80
```

would draw a small square.

3. RENUMBER has been added. The syntax for this is:

```
RENUMBER [line-range] [,STEP line-no.] [,AT line-no.]
```

The line-range specifies the portion of the program to brenumbered in a similar way to LIST and DELETE. The STEP value is the increment used for the renumbering (default 10) and the AT value is the new line number for the first line that will be renumbered. AT defaults to the existing line number of the first line specified by the line-range, or to the existing first line number of the program if the line-range specifies from the start of the program.

Renumbering a program clears all variables, and the RENUMBER command must be the last command on the line.

A 'No memory' error will be given if there is not enough free RAM available to renumber the program (GOTO 1, for example, requires three bytes less memory than GOTO 1000). A 'Cannot RENUMBER' error will be given and the renumber aborted (leaving the program unchanged) if the program cannot be renumbered for some reason. Note that RENUMBER uses extra work space RAM, the amount used depending on the number of lines in the program (four bytes per line). Therefore if a 'no memory' error occurs, then it may be possible to renumber the program in several sections.

If an error occurs as a result of invalid RENUMBER parameters, then the RENUMBER command is listed as the erroneous line (eg. the last new line number would exceed the maximum line number of 9999 or the new line number of the last line in the line-range would overlap the next program line). If, however, a program line is listed as the erroneous line, then the fault lies with the indicated program line (eg. a GOTO specifying a line that does not exist, or a line that would exceed the maximum length of 251 characters when renumbered).

A few commands cannot be renumbered. These are: AUTO, DELETE, LIST, LLIST and RENUMBER.

4. AUTO has been added. The syntax for this is:

AUTO [STEP line-no.] [,AT lin-no.]

The STEP value is the increment used for the new lines (default 10) and the AT value is the first new line to enter (default 10). For each new line that AUTO creates, the line number is displayed on the screen and the cursor placed just after it. The text for the line can then be typed in, the line being entered when 'return' is pressed in the normal way. If the line already exists, then instead of just printing the line number, the entire line will be listed, and the cursor placed at the end. The existing line can then be edited, or deleted if not required. If the line number that AUTO automatically printed is edited, then the next line number that is printed will be the new line number plus the STEP value.

AUTO mode will be exited if a line without a line number is entered, if the next line number to be printed exceeds the maximum line number of 9999, if CTRL-ESC is pressed or if some other error occurs.

5. 10 function keys have been added to the keyboard driver, with a total of 100 bytes available for the strings programmed into them. The memory for the function keys is permanently allocated from HIMEM when the computer is cold reset. The number of bytes used for each function key string is the number of characters in the string plus one.

Currently, nearly all SHIFT, CTRL and ALT sequences produce useful character codes from the keyboard, and there are no useful 'gaps' in the keyboard map to allow any of these three keys to be used for function keys. Instead, the LOCK key is used. The use of the LOCK key remains compatible with it's use in pre-2.0 versions, ie. pressing SHIFT, CTRL or ALT and the LOCK key will enter SHIFT, CAPS or ALT lock mode, and pressing then releasing the LOCK key will return to normal unlocked mode. n addition, using the LOCK key in the same way as the SHIFT, CTRL and ALT keys with one of the number keys (ie. pressing the 2 keys simultaneously) will act as a function key. This can be done in normal or CAPS lock mode without affecting the current LOCK mode.

From BASIC, a function key is programmed with the FKEY statement. It's syntax is:

FKEY numeric-expression , string-expression

The function key specified by the numeric-expression, which must be in the range 0 to 9, will be programmed with the string-expression, provided that enough free function key RAM is available. If there is insufficient RAM, then an error will be returned and the current setting of the function key will be unaffected. If the string is a null string, then pressing the function key will have no effect. This is the initial state of all the function keys.

To program BASIC commands (such as LIST), it may be required to end the string with a carriage-return, so that just a single key press will perform the command. This can be done by appending ' & CHR$(13) ' to the end of the string-expression.

To support the FKEY statement, a new keyboard driver function is available, as follows:

@FKEY (keyboard function 00000100B)

Input: DE -> new function key string.
C = function key number, 0 to 9.

Output: A = error code, 0 if successful.
DE -> current function key string.

The function key specified by the number in C will be programmed with the string pointed to by DE. The first byte of this string is the length byte. DE is returned pointing to the function key string, which will either be the string that was programmed if A=0, or the old string if an error occurred. Thus if a string with a large length byte is passed (eg. 0FFh) so that an error always occurs, then the current setting can be obtained.

6. Error trapping and handling statements and functions have been added to BASIC. The syntax of these are:

ON EXCEPTION GOTO line-number

When an error occurs from a program line, control will be transferred to the specified line-number, as in an ordinary GOTO. If the line-number is 0, then any existing error trapping will be forgotten and no error trapping will occur.

When the error routine is entered, error trapping will initially be turned off, although another ON ERROR statement may be given. When the main program is returned to with CONTINUE, the error trapping will be restored to the original line number.

EXCEPTION numeric-expression [, line-number]

Causes an error, the number of which is specified by the numeric-expression. The error will appear to come from the line specified by the line-number, or from the current line if the line-number is not specified. Error numbers are listed in appendix A.

ERRLIN, ERRNUM

These two functions return the line number and the error number respectively of the last error that occurred.

CONTINUE [line-number]

In pre-2.0 versions, CONTINUE did nothing when used in a program, and continued execution of a stopped program when used in immediate mode. The latter use remains unchanged in version 2.0, except that it now remains valid to continue the program after errors have occurred. Only changing the program (including with RENUMBER) stops continue being valid.

When used in a program, CONTINUE with no parameters causes the exception routine to be exited and the statement that caused the error to be re-executed. If [illegible] line.

The error numbers used are the same as those in pre-2.0 versions, but with some additions. When an error above 128 occurs, it is assumed to be an operating system error, and the error message printed is:

*** System error n

as in previous versions. Errors below 128 are considered to be BASIC errors, and either an error message is printed, or the message:

*** BASIC error n

In previous versions unrecognized error numbers below 128 simply printed 'BASIC corrupted'. This can no longer occur.

Appendix A lists the error numbers and error messages.

7. BASIC now calls a hook in RAM called BASEXT (see appendix B) when an unrecognized command is given. By default, this will just return, but may be changed to jump elsewhere to allow extension commands and statements to be executed. If the statement is still not recognized, then any patched-in routine should just return. If the statement was executed successfully, then one address should be removed from the stack (POP HL or similar) and then a return done with HL' pointing to the first item after the statement (ie. the end of the line, a ! comment or a : statement separator). It is possible to access some useful internal routines by performing a RST instruction followed by a series of single-byte function codes. For example, an expression can be evaluated. Another RST allows errors to be generated. See appendix C for details of these and appendix D for an example BASIC extension.

8. The printer driver now calls a hook in RAM called PRNDEF (see appendix B) with the character to be printed in C and B=FFh. By default, this willjust return, but may be changed to jump elsewhere to allow the character to be changed. If C<>FFh is returned, then C is printed. Similarly B will then be printed if it is not equal to FFh. In this way, characters may be transformed into other characters or into another character pair.

   A similar hook called SERDEF exists for the serial driver, providing the same facilities for users with serial printers.

9. BASIC now calls a hook in RAM called STRCMP (see appendix B) after comparing each character during string comparisons. By default, this just returns but may be changed to jump elsewhere to allow the comparison result to be changed. One character is in A and another at (HL), so the instruction CP (HL) will set the flags correctly for normal ASCII characters.

10. In pre-2.0 versions, there were two kinds of reset in the system: warm and cold. A cold reset completely reset MOPS and entered the cartridge if present, or BASIC otherwise. A warm reset reset MOPS and the device drivers and re-entered the cartridge or BASIC, allowing it to start up again but without initializing all it's variables etc.

In version 2.0, a cold reset has the same effect, and either the cartridge or BASIC will start up depending on whether or not a cartridge is present. Once the system has started, however, MOPS can be warm reset either in the normal way (by pressing the reset button) or by deliberate action by the cartridge or BASIC.

The variable CENBLE (see appendix B) allows the cartridge to be enabled or disabled, and it's value is preserved throughout a warm reset of MOPS. Normally it is zero, which causes the traditional behaviour ie. BASIC is started unless there is a cartridge, in which case it is always started instead. If CENBLE is non-zero, then on any sort of MOPS warm reset the cartridge is never entered, and is thus effectively disabled. Thus a cartridge can set CENBLE to non-zero and warm reset MOPS, which will cause BASIC to be entered. The cartridge will only be entered again if a cold reset occurs, or if CENBLE is set back to zero and MOPS warm reset from BASIC.

In pre-2.0 versions, the MOPS variable WARM_FLAG was used to decide whether a cold or a warm reset was in progress. In version 2.0, it is still used for the same purpose until after MOPS has been initialized. Then, if MOPS was warm reset, the variable RESTART (see appendix B) is copied to WARM_FLAG. Most of the time, RESTART is non-zero so that when the reset button is pressed, MOPS does a warm reset (WARM_FLAG <> 0) and then another non-zero value is copied to WARM_FLAG before the cartridge or BASIC is re-entered. However, if MOPS is warm started by the cartridge (or BASIC) then RESTART can be set to zero, causing a warm reset of MOPS (WARM_FLAG <>0) but a cold reset of the cartridge or BASIC (WARM_FLAG now = 0).

When the cartridge or BASIC is entered with WARM_START=0, the value of the variable MOPS_WARM (see appendix B) can be examined if required to determine whether MOPS did a cold or a warm reset. WARM_FLAG is copied to MOPS_WARM immediately before RESTART is copied to WARM_FLAG. MOPS devices and expansion cards are reset before this, and perform the same sort of reset as MOPS. An expansion card ROM can thus find out whether or not the cartridge or BASIC will be cold or warm reset by looking at the value in RESTART. For example, the built-in keyboard device, which provides function keys, needs to allocate RAM when MOPS is cold reset but not when warm reset. When MOPS is warm reset, it needs to clear any existing function key definitions if the cartridge or BASIC cold reset, but not if warm reset. Similarly other expansion card ROMs may need to allocate RAM on a cold reset, and only clear it when a possible change of application program takes place (ie. when the cartridge or BASIC is cold reset).

Thus by using CENBLE and RESTART carefully, a cartridge program can start up BASIC, and a BASIC program can start up the cartridge without completely cold resetting the system.

When a cartridge starts up after a cold reset, it may decide that it wants to start BASIC up instead (eg. the VT-DOS cartridge does not start up if there is no VT-DOS disk controller card also plugged in). Rather than setting CENBLE and completely resetting MOPS in the manner described above, it can also jump to the address obtained by subtracting 22 from the address in DE when the cartridge was first entered. The MOPS ROM must be in page 0 and interrupts disabled. This works for all past versions of MOPS, and is the method used by the VT-DOS cartridge. This method is only recommended as a way for a cartridge to return immediately to MOPS; as a general method of starting up BASIC from a cartridge it is not really adequate since MOPS may not then be in a freshly reset state. Of course, if a warm reset occurs then the cartridge will be re-entered unless CENBLE was set to non-zero, even if it was BASIC that was running.

The actual method the cartridge uses to restart MOPS is to jump to a hook in RAM called RESET (see appendix B). This simply zeros the RESTART variable and resets MOPS. Before jumping to this hook the variable CENBLE must be set up in the required state.

When a cold start of the application takes place (whether it was a warm or cold reset of MOPS), an expansion ROM or a cartridge can give BASIC a program to execute. This is done by copying the BASIC pogram to BASIC's normal program start address (19EFH) and setting various flags in the variable BASFLG (see appendix B). The program copied into memory must be exactly as written out by BASIC's SAVE command, but without the 16 byte header that BASIC writes out at the start of the file. Note also that disk files with an extension of .CAS have an extra 128 byte header that also must not be included. Note that it is possible to produce an expansion ROM or cartridge that puts a BASIC program into memory as above, whilst still remaining compatible with pre-2.0 versions. The effect will simply be that the older versions of BASIC will ignore the program.

There are four flags in BASFLG, as follows:

- Bit 0 - If this bit is set, then when BASIC starts up the initial flashing 'VIDEOTON' screen will be ommitted.
- Bit 1 - If this bit is set, then when BASIC starts up the 'copyright' sign-on message and the number of bytes free will not be printed.
- Bit 2 - If this bit is set, then when BASIC starts up it will not automatically perform a 'NEW', thus allowing BASIC to recognize that it has a program in memory.
- Bit 3 - If this bit is set, then when BASIC starts up it will automatically perform a 'RUN'. This is only useful if bit 2 is also set.

When BASFLG is set up, the whole byte should be set up. It is not sufficient to just set the required bits.

11. Address C30Dh in the main ROM has been fixed and can be jumped to in order to start up the cartridge. This is for compatibility with past versions.

12. The HEX$(x) function (where 0 <= x < 65536) returns a four-character string hich contains the hexadecimal representation of x in upper-case ASCII. x and HEX$(x) are treated as unsigned 16 bit integers.

    The DEC(x$) function returns a number which is the decimal equivalent of the hexadecimal number in x$, which must consist of ASCII digits and upper or lower case letters in the range 'A' to 'F' or 'a' to 'f' respectively. x$ and DEC(x$) are treated as unsigned 16 bit integers.

13. The variable BLINK (see appendix B) can be set to non-zero before calling the keyboard CHIN function in order to obtain a flashing cursor. The keyboard call does not change the value of BLINK, but an editor CHIN call sets it back to zero. Note that BLINK can be POKEd to 255 from BASIC before an INKEY$ call to obtain a flashing cursor.

The 2.0 error numbers and the messages corresponding to BASIC error numbers are listed below.

FFH to E5H

The meanings of these errors are unchanged from 1.x.

.FKEY - E4H

An invalid function key number was specified, when programming a function key, or the new function key string would cause the total number of bytes used for function keys to exceed 100.

.POLY - E3H

Too many sides were specified when drawing a polygon. The maximum is 20.

E2H - D0H

Not currently used.

BFH - 80H

Reserved for VT-DOS.

7FH - 12H

Not currently used.

.BADREN - 11H

"Cannot RENUMBER". A renumber command was given but would result in an invalid program egline numbers out of sequence, lines greater than 251 bytes in length, or GOTOs to non-existent lines.

.BADFIL - 10H

"Bad file".

.VARDEC - 0FH

"Variable declared twice".

.MISMAT - 0EH

"Type mismatch".

.OVRFLW - 0DH

"Overflow".

.CANTRD - 0CH

"Cannot READ".

.DIV0 - 0BH

"Cannot divide by 0".

.CANTCNT - 0AH

"Cannot CONTINUE".

.NOGOSUB - 9

"No GOSUB".

.NOFOR - 8

"No FOR".

.NODATA - 7

"No DATA".

.NOMEM - 6

"No memory".

.BADSUB - 5

"Bad subscript".

.BADARG - 4

"Bad argument".

.NOARG - 3"

"Argument missing".

.NOLINE - 2

"Line missing".

.SYNTER - 1

"Not understood".

In version 2.0, no previously published or unpublished variables or data areas have been moved. Several new variables and data areas were required, however, and these reside in RAM that in previous versions was not used. These areas are detailed below.

In addition, 100 bytes is used for storage of function key strings. This is taken at cold reset time from HIMEM.

The areas of RAM used are:

```
     700H - 724H   - Miscellaneous  new BASIC variables used
                     for  the  new error trapping  statements
                     and the RENUMBER and AUTO commands.
     725H  PRNDEF  - Printer  character translation  hook.  A
                     jump can be placed here to hook in a new
                     routine.
     728H  SERDEF  - Serial character translation hook.  Used
                     for  the  same purpose as  PRNDEF  above
                     when a serial printer is used.
     72BH  RESET   - Jumping  to here will start up BASIC  or
                     the  cartridge,   depending  on  whether
                     CENBLE  is zero or non-zero.  It is  not
                     actually a hook as it is not anticipated
                     that  another routine will want to  hook
                     itself in.
     72EH  STRCMP  - BASIC string comparison hook. A jump can
                     be  placed  here to a routine  that  re-
                     defines  the character order for  string
                     comparisons.
     731H  BASEXT  - BASIC extension statement hook.  A  jump
                     can  be placed here to jump to a routine
                     to implement extra statements in BASIC.
     734H - 740H   - Used internally.
```

The RAM above was reserved in previous versions for a 25th screen line in the ASCII MAP, but was not actually used. In order to attach a routin to one of the above hooks, the three bytes at the hook should be saved in memory somewhere. Then a three byte jump to the new routine is placed in the three hook bytes. When this new routine is executed, it should generally end by executing the OLD contents of the hook. This ensures that if more than one routine is hooked in they all chain together, with the latest routine getting priority.

```
     E35H - E3AH   - Used internally.
     E3BH  BLINK   - If non-zero before a keyboard CHIN call,
                     causes the cursor to blink.  Reset by an
                     editor CHIN call.
```

```
E3CH RESTART - Non-zero for ordinary warm reset, zero
               to warm reset MOPS, cold reset
               application.
E3DH MOPS_WARM-Type of reset that MOPS did (warm or
               cold) when application is cold started.
E3EH CENBLE  - Zero to enable cartridge, non-zero to
               disable it.
E3FH BASFLG  - BASIC start up flags.
               Bit 0 - skip flashing VIDEOTON screen.
               Bit 1 - skip copyright sign-on message.
               Bit 2 - don't do a NEW on start up.
               Bit 3 - Do an automatic RUN on start up.
E40H VERSION - MOPS version number. High nibble is the
               major version number, low nibble is the
               minor version number, ie. 20H for
               version 2.0. Zero for all versions prior
               to version 2.0.
E41H - E47H  - Not currently used.
```

The RAM above was reserved in previous versions for use by the cassette driver, but was not actually used.

```
16ACH - 17FFH - Used internally by BASIC. This area,
                which is between USER_BASE and the start
                of BASIC's variables, was not used in
                previous versions.
```

BASIC 2.0 allows extended statements to be linked into it by hooking into the BASEXT hook.

It is important to note that any extended statement routines hooked in in this way must be compatible with BASIC's use of the Z80 registers. This has been documented previously, but generally is as follows:

- IX - points to the start of BASIC's variables.
- IY - points to the last used byte on BASIC's stack (this is not the same as the Z80 stack).
- HL'- points to the next syntactic item in the program source.
- B' - contains the current syntactic item token from the program source.

It is not generally necessary to access any of these registers, and they should not be arbitrarily changed. Instead, they are maintained and updated by many of the BASIC functions available (see below), particularly the GET function.

Once an extension statement routine has hooked itself into BASIC, it will get control whenever an unrecognized command or statement is entered (including when an implied LET statement is entered, such as A=1 instead of LET A=1). HL' will then point to the first character of the statement, which will be tokenized. If the statement at HL' is not recognized, then the contents of the old hook should be executed with HL' preserved.

If the statement at HL' is recognized, then any expected parameters should be evaluated and the statement acted upon as appropriate. Then the BASIC return address should be removed from the stack (with a POP HL or similar) and a RET executed. HL' and B' should then have been updated beyond the statement and it's parameters, and will thus point to a colon, a ! comment or the end of the line if the syntax is correct. It it does not point to one of these, then BASIC will generate a 'Not understood' error.

It is useful for an extension statement to be able to access certain internal BASIC routines, such as numeric and string expressions. These and many other routines can be accessed via the BASIC function restart (see below). Other restart (RST) routines are available, as follows:

RST 8

Generate a BASIC error. The error number must be contained in the byte immediately following the RST 8 opcode. BASIC's error routine will be entered, and any error trapping set up will come into effect. If no error trapping is in effect, then the appropriate error message will be printed. The routine performing the RST is never returned to. This is available in all previous versions of BASIC too.

RST 10H

Check for a MOPS error. This RST can be performed immediately after a MOPS call, and will either immediately return (if the Z flag is set) or will enter BASIC's error routine (see RST 8 above) with the error code in A. This is available in all previous versions of BASIC too.

RST 18H

Perform BASIC function. Following the RST 18H opcode should be a list of bytes, each byte specifying a BASIC function to perform. The functions are then performed in sequence. The last function number in the list should have the top bit set, which indicates that the program continues in the next byte.

The functions available are as follows. Only the first 15 are available in previous versions of BASIC; the rest are new in 2.0. Where functions use numbers and strings on BASIC's stack, no check is made for items of the correct type being on the stack; it is up to the caller toensure this. IX, IY, HL' DE' and B' are preserved unless otherwise stated. Other registers are corrupted unless otherwise stated.

FADD - 0

Floating point addition. The top two numbers on BASIC's stack are added together, and are replaced by the result. AF only is preserved.

FDIV - 1

Floating point division. The top number on BASIC's stack is divided into the second number on the stack, the result replacing them both. AF only is preserved, apart from the carry which may be corrupted.

FMULT - 2

Floating point multiplication. The top two numbers on BASIC's stack are multiplied together, the result replacing them both. AF only is preserved.

FSUB - 3

Floating point subtraction. The top number on BASIC's stack is subtracted from the second number, the result replacing them both. AF only is preserved.

FNEG - 4

Floating point negation. The top number on the stack is negated. BC, DE and HL only are preserved.

5

Used internally.

XPUSH - 6

The number in floating point variable X is pushed onto BASIC's stack.

YPUSH - 7

The number in floating point variable Y is pushed onto BASIC's stack.

XASSIGN - 8

The number on top of BASIC's stack is copied to floating point variable X, but is not removed from the stack.

YASSIGN - 9

The number on top of BASIC's stack is copied to floating point variable Y, but is not removed from the stack.

XPOP - 0AH

The number on top of BASIC's stack is copied to floating point variable X and is removed from the stack.

YPOP - 0BH

The number on top of BASIC's stack is copied to floating point variable Y and is removed from the stack.

FDUP - 0CH

The number on top of BASIC's stack is duplicated, so that two equal numbers are on the stack.

FPOP - 0DH

The number on top of BASIC's stack is copied to the 6 byte variable pointed to by HL, and is removed from the stack.

PNUM - 0EH

The syntax:

(numeric-expression)

is parsed, the expression evaluated and the result pushed on top of BASIC's stack. HL' and B' are updated appropriately.

FDROP - 0FH

The number on top of BASIC's stack is removed, and it's value ignored.

VPUSH - 10H

The number in floating point variable V is pushed onto BASIC's stack.

WPUSH - 11H

The number in floating point variable W is pushed onto BASIC's stack.

VASSIGN - 12H

The number on top of BASIC's stack is copied to floating point variable V, but is not removed from the stack.

Copyright (1987) IS Systems Ltd.

WASSIGN - 13H

The number on top of BASIC's stack is copied to floating point variable W, but is not removed from the stack.

VPOP - 14H

The number on top of BASIC's stack is copied to floating point variable V and is removed from the stack.

WFQPOP 15H5H

The number on top of BASIC's stack is copied to floating point variable W and is removed from the stack.

SIN - 16H

The number on top of BASIC's stack is replaced by the sine of that number.

COS - 17H

The number on top of BASIC's stack is replaced by the cosine of that number.

ATN - 18H

The number on top of BASIC's stack is replaced by the arctangent of that number.

TAN - 19H

The number on top of BASIC's stack is replaced by the tangent of that number.

EXP - 1AH

The number on top of BASIC's stack is replaced by the constant e raised to the the power of that number.

INT - 1BH

The number on top of BASIC's stack is replaced by the value of the largest integer not greater than that number.

LOG - 1CH

The number on top of BASIC's stack is replaced by the natural logarithm of that number.

SQR - 1DH

The number on top of BASIC's stack is replaced by the positive square root of that number.

INVOL - 1EH

The second number on BASIC's stack is raised to the power of the top number, the result replacing them both.

FCMP - 1FH

The top two numbers on BASIC's stack are compared and removed from the stack, and the result returned in the flags register with A consistent with the flags. The M condition is true if the top number was greater than the second number, the Z condition is true if the numbers were equal, and the P condition is true (and Z false) if the top number was less then the second number.

FCPL - 20H

The top number on BASIC's stack is replaced by the ten's complement of that number (ie. the mantissa digits are subtracted from 0).

GETDIG - 21H

The value of a single decimal digit is returned from the top number on BASIC's stack. The number is not removed. The most significant digit is numbered 4, and the number increases by one for the next significant digit. The digit number is passed in B and the digit value is returned in A. HL, DE and BC are preserved.

PUTDIG - 22H

A single digit is written into the top number on BASIC's stack. The number is not removed. The digit number is passed in B and is as in GETDIG above. The new value, which must be in the range 0 to 9, is passed in A. HL, DE and BC are preserved.

FNORM - 23H

The number on top of BASIC's stack is shifted left or right by nibbles until the most significant digit is in the normal position (digit position number 4, see GETDIG and PUTDIG above). Digits 2 and 3 are taken into account.

FZERO - 24H

The digits of the number on top of BASIC's stack are tested for being all zero, and if they are then the exponent byte is set to zero to ensure that a zero number has the correct representation for zero. The Z condition is true if the number is zero.

FOUT - 25H

The number on top of BASIC's stack is turned into it's free format ASCII representation. The number is not removed from the stack. HL is returned pointing to the first byte of the number, which is in an internal BASIC buffer.

PUSH - 26H

The signed number in HL is turned into a floating point number and pushed on to BASIC's stack.

POP 27H

~~POPOP 2707H~~

The number on top of BASIC's stack is turned into a signed 16 bit integer and returned in HL. The number is removed from the stack.

$PUSH - 28H

The string pointed to by HL+1 is pushed onto BASIC's stack. The first byte of the string is assumed to be the length byte. The byte pointed to by HL is ignored.

$POP - 29H

The string on top of BASIC's stack is copied to memory, and removed from the stack. HL must point to the area of memory for the string, and the byte pointed to by HL must contain maximum number of bytes that can be accommodated, including the length byte of the string.

$CMP - 2AH

The top two strings on BASIC's stack are compared. The strings are removed from the stack and the results are returned in AF as for the FCMP function. The STRCMP hook is called for each character compared.

FEXPR - 2BH

A numeric-expression is parsed and evaluated, the result being pushed onto the top of BASIC's stack. HL' and B' are updated appropriately.

$EXPR - 2CH

A string expression is parsed and evaluated, the result being pushed onto the top of BASIC's stack. HL' and B' are updated appropriately.

GET - 2DH

The next syntactic item in the program (as pointed to by HL') is obtained and returned in B' and DE'. Various flags describing the item are returned in AF. HL' is updated appropriately.

If the item is a token, then this is returned in A (and will be the same as the byte in B'). Before returning, the token will be compared with the token for a colon. Thus the NC condition indicates the end of a statement (colon, ! comment or end of line). The Z condition indicates a colon. Token values have been documented previously, and new values are as follows:

CFH - AUTO
CEH - FKEY
CDH - RENUMBER
CCH - EXCEPTION
ADH - ELLIPSE
ACH - RECTANGLE
ABH - POLYGON

If the item obtained is not a token, then A (and B') contain other flags, as follows:

Bit 0 - Set if a numeric or string identifier, otherwise a numeric or string literal item.
Bit 1 - Set if a numeric item, else a string item.

If the item is a literal, then it is pushed onto the top of BASIC's stack. It should not however be used in this form; the EXPR or $EXPR should be called first.

If the item is an identifier, then DE' is returned pointing to the first data byte of the symbol table entry. C' will contain the symbol table type byte.

LIST - 2EH

The tokenized string pointed to by HL is expanded and printed. The end of the string is indicated by an FFH byte, and HL is returned pointing to the byte following this. Note that this is the format in which programs are stored, and if this function is invoked with HL pointing passed the line number field of a program line, then the program line will be listed (without the line number).

Several example programs are given here to illustrate the use of some of the new features of BASIC 2.0.

Extension statement

This example implements a genuinely useful example extension statement for BASIC. Normally, the serial baud rate is set by POKEing a variable with a number which must be looked up in table of baud rates. The serial format is even more inconvenient, and requires the various parameters (parity, number of data bits, number of stop bits) to be encoded into a byte and then POKEd into another variable. Finally a third variable has to be POKEd to tell the serial driver that the parameters have been changed.

The program below implements a BASIC statement called BAUD. This allows the baud rate and format to be set up in an easily remembered format.

The program is in an assembler listing form, and this given details of the use and syntax of the statement. Following this a short BASIC program is given which POKEs the machine code into memory.

String Comparison

This program allows the order of the character codes to be changed for the purposes of string comparison in BASIC. It simply swaps the order of the codes for A and B, so that the string "B" is less than the string "A".

The program is in an assembler listing form, and a BASIC program is also given which POKEs the machine code into memory.

BASIC Programs in ROM

These two programs are fairly similar and allow BASIC programs to be blown into ROM. The first program is for a cartridge ROM, and the second for an expansion ROM.

Printer Re-definition

This program illustrates the use of the PRNDEF hook. No attempt is made to implement a useful translation; this example program simply converts all space characters into the sequence ' <> '.

The program is in an assembler listing form, and a BASIC program is also given which POKEs the machine code into memory.

Copyright (1987) IS Systems Ltd.